SELFWATTS: On-the-fly Selection of Performance Events to Optimize Software-defined Power Meters

Guillaume Fieni, Romain Rouvoy, Lionel Seinturier

SELFWATTS: On-the-fly Selection of Performance Events to Optimize Software-defined Power Meters

Guillaume Fieni, Romain Rouvoy*, Lionel Seiturier
*Univ. Lille / Inria / *IUF*
firstname.lastname@univ-lille.fr

***Abstract*—Fine-grained power monitoring of software-defined infrastructures is unavoidable to maximize the power usage efficiency of data centers. However, the design of the underlying power models that estimate the power consumption of the monitored software components keeps being a long and fragile process that remains tightly coupled to the host machine and prevents a wider adoption by the industry beyond the rich literature on this topic.**

To overcome these limitations, this paper introduces SELFWATTS: a lightweight power monitoring system that explores and selects the relevant performance events to automatically optimize the power models to the underlying architecture. Unlike state-of-the-art techniques, SELFWATTS does not require any *a priori* training phase or specific hardware to configure the power models and can be deployed on a wide range of machines, including heterogeneous environments.

I. INTRODUCTION

Modern data centers are continuously trying to maximize the *power usage efficiency* (PUE) of their hardware and software infrastructures to reduce their operating cost and eventually their carbon emission. In the context of cloud operators, this PUE optimization process requires more and more to carefully and deeply understand the implication of complex and continuously-evolving software infrastructures that are made of hundreds of software services deployed across the physical infrastructure.

While physical power meters, like IPMI, offer a suitable solution to monitor the power consumption of physical servers, they fail to support the energy profiling at a finer granularity—*i.e.*, dealing with the software services that are distributed across such infrastructures. To address this concern, software-defined power meters have therefore been introduced as a solution to support process-level power estimations, through the implementation of dedicated power models that leverage system-level metrics to estimate the power consumption at the granularity of software services [1], [2]. The design of the underlying power models that estimate the power consumption of the monitored software components keeps being a long and fragile process that remains tightly coupled to the host machine [3], [4]. Furthermore, the deployment of such software-defined power meters remains a critical issue when facing the diversity of hardware settings in the wild, which prevents a wider adoption by the industry. In particular, the proposed power models either cannot be exploited because of the unavailability of required metrics or, at best, deliver incorrect power estimations of hosted software services [5], [6].

This key limitation, therefore, calls for more adaptive approaches that can adjust and optimize the power model to the hardware constraints of a given deployment target. More specifically, this paper addresses the self-optimization of power models by *i)* automatically selecting the most relevant set of hardware performance events and *ii)* continuously inferring the best power model for the target architecture. The proposed approach ensures that our self-optimized power models keep delivering accurate power estimations while fitting to the hardware constraints imposed by the deployment. Our approach is made available as a software solution, named SELFWATTS, that can be quickly deployed at the scale of a data center to monitor the power consumption of software containers or *virtual machines* (VM), with negligible overhead. Once deployed, SELFWATTS keeps exploring the space of available hardware performance events to detect if unexplored events contribute more favorably to the accuracy of the power model.

Interestingly, we show that SELFWATTS delivers real-time power estimations that compete with the state-of-the-art software-defined power models while offering a plug-and-play solution to data center administrator for monitoring the power consumption of their infrastructure services, as well as reporting the energy consumption of their customers, no matter their hardware constraints. This contribution thus paves the way for more sustainable cloud services by exposing this key performance indicator to interested stakeholders (*e.g.*, cloud administrators and customers) and encourage them to reduce their environmental footprint.

In the remainder of this paper, we start by providing some background on state-of-the-art power models and their limitations (cf. Section II) prior to introducing our contribution (cf. Section III). Then, we detail the implementation of SELFWATTS as an extension of the SMARTWATTS software-defined power-meter (cf. Section IV) and we assess its validity on three scenarios (cf. Section V). We conclude in Section VI.

II. RELATED WORK

Before introducing SELFWATTS, we report on the limitations of the state of the art in this domain, starting from energy measurement methods (cf. Section II-A) and then focusing on the challenge of feature selection for software power models (cf. Section II-B).

A. Energy Measurement Methods

Over the years, hardware and software power meters have evolved to deliver hardware-level power measurements with different levels of granularity, from physical machines to electronic components and running software.

RAPL [7] exposes additional *hardware performance counters* (HWPC) to report on the energy consumption of the CPU since the "Sandy Bridge" micro-architecture for Intel (2011) and "Zen" for AMD (2017). RAPL divides the system into domains (PP0, PP1, PKG, DRAM) that report the energy consumption according to the requested context. The PP0 domain represents the core activity of the processor (cores + caches), PP1 the uncore activities (LLC, integrated graphics, etc.), PKG aggregates PP0 and PP1, and DRAM covers the DRAM energy consumption. The RAPL feature does not require any hardware modification, but the list of available domains depends on the model of the CPU. Desrochers *et al.* demonstrate the accuracy of the DRAM power estimations of RAPL, especially on Intel Xeon processors [8]. Thus, even if RAPL does not capture the software-level energy consumption, we believe it offers a relevant ground truth for modeling the power consumption at the scale of the processor.

POWER CONTAINERS [9] proposes to account for and control the power and energy usage of individual requests in multicore servers. However, the deployment of power containers requires to pre-calibrate the power model with offline samples and then recalibrate these power models with online context samples. This implies that several micro-benchmarks require to be executed to infer the coefficients of the power model, thus imposing a long delay that prevents it to be deployed in production when multiple generations and models of machines are available.

BITWATTS [10] is a monitoring middleware providing real-time power estimations of software processes running at any level of virtualization in a system. BITWATTS includes a power model that computes a polynomial regression for each frequency supported by the CPU (including Turbo Boost frequencies). BITWATTS requires a physical power meter (PowerSpy) with a manual calibration phase to benchmark every frequency supported by the CPU. Unfortunately, this procedure also prevents BITWATTS to be deployed on a wide panel of hardware architectures. WATTSKIT [2], which is an extension of BITWATTS, supports the power monitoring of distributed services, but suffers from the same limitation when it comes to the effective deployment of the solution in a cluster of heterogeneous machines.

WATTWATCHER [11] is a tool that can characterize the energy consumption of a workload. To do so, the authors combine several calibration phases to train a power model that fits a CPU architecture. The authors propose a special power model generator that can target any CPU architecture, but requires to be carefully described *a priori*. Unfortunately, this power model uses a predefined set of HwPC events as input parameters, which may not be available on some CPU architectures, thus preventing the exploitation of the generated power model.

SMARTWATTS [1] offers a self-calibrating software power meter for container-based environments. SMARTWATTS can deliver real-time power estimations about CPU and DRAM for the software containers, and power models are calibrated online, which saves the expensive calibration phase. Again, like WATTWATCHER and previous approaches, the HwPC events remain to be fixed at startup, which requires to know which ones to use for a target architecture.

B. Feature Selection for Software Power Models

As mentioned above, power models are learned from raw metrics that are expected to relate the activity of the hardware components energy consumption. Modern CPUs provide sverals *Performance Monitoring Units* (PMU), which implement a limited number of HwPC slots that can be used to monitor a large number of performance events. In the literature, the selection of the relevant performance events that feed a power model is mostly achieved offline, thus requiring a calibration phase during which a target machine has to execute several workloads over a long period to identify such relevant raw metrics.

To select the key metrics, the literature usually builds on *Pearson* or *Spearman* correlation [1], [4] and *Principal Component Analysis* (PCA) [12], which often leads to consider performance events like *unhalted core cycles*, *unhalted ref cycles*, *instructions retired*, *llc misses/prefetch*, or *memory transactions cycles* [1], [4], [9]–[11]. Yet, this selection phase requires to be executed on every single target architecture to make sure that the relevant performance events are made available, which inevitably impacts the cost and the scale of the deployment of software-defined power meters. Furthermore, all these *a priori* calibration phases share the same limitation: the selected performance events highly depend on the nature of the calibration workload, which may strongly differ from the workload monitored in production and thus question the accuracy of the resulting power model.

C. Hardware Power Optimizations

This contextual issue is particularly challenging to capture as modern processors embed several mechanisms that are autonomously triggered in order to optimize the idle power consumption and the performance of the host machine upon context. For example, Intel CPUs are currently implementing the following power-aware hardware features:

P-states are *performance power states*, where each state specifies the voltage and the clock frequency at which the CPU operates. This allows the CPU to maintain performance objectives while minimizing power consumption. The operating system picks the most suitable state according to the current usage of the processor by the running workload. The number of supported P-states depends on the micro-architecture and the model of the CPU. Currently, the highest state is `P0` when the CPU operates at the highest voltage and frequency, leading to an increase in performance along with the dissipated heat.

C-states are *idle power states*, also known as core C-states (CC-states), package C-states (PC-states) and logical C-states, specify parts of the CPU that can be powered down to reduce the power consumption depending on usage and the latency cost imposed by power states transitions. The highest state is `C0` when the CPU is fully operational, and the lowest is `C8` when the CPU is inactive and its state saved to LLC before its power cut-off by the power gate transistors. The number of supported C-states also depends on the micro-architecture and model of the CPU. While the operating system suggests to the *Power Control Unit* (PCU) a target state for each core based on the current load of the machine, the PCU autonomously decides the most suitable core and package C-state to optimize the power consumption of the CPU.

Turbo Boost feature allows the CPU to run one or many cores to higher P-states than usual, leading to an increase in performance for a short period. However, this feature becomes only available when the CPU is operating below its rated maximum temperature, current, and power limits. This mode leverages the C-states as the frequency depends on the number of active cores—*i.e.*, the more cores in idle states the higher frequency of active cores. Typically, turbo boost becomes active when the system requests a transition to the P-state `P0`, the operating conditions are below certain model-specific limits and the workload demands more performance. Several turbo frequencies are available depending of the model of the CPU and the current workload. Additionally, while standard workloads can use all turbo frequencies, the AVX2 and AVX512 workloads have dedicated turbo frequencies.

CKE-states are memory power-down modes that allow DIMM ranks to powered off dynamically when unused. These states are linked to the package C-states where the deepest states allow the memory to enter the self-refresh mode to greatly reduce its power consumption. An IDLE counter is available for each memory rank and determines its CKE mode to maximize the opportunities to power-off unused ranks even under memory-intensive workloads. The *integrated Memory Controller* (iMC) can autonomouslypower down the DIMM ranks to save energy at the cost of more latency when it will be woken up.

When these mechanisms are triggered by the CPU or the DRAM components, some HWPC events are no longer correlated with the power consumption. In the literature, some power models are *Dynamic voltage and frequency scaling* (DVFS) aware, including the *Turbo Boost—i.e.*, P-states effects on the power consumption of the CPU [10]. However, to the best of our knowledge, none of the existing power models take into account the power states in their power models, which leads to incorrect power estimations, when the CPU enters such states, thus preventing their adoption at scale. This limitation is mainly due to the emergence of new energy optimizations, such as the migration from the legacy generic ACPI CPU performance scaling driver to processor-driven ones, such as Intel P-states which can even offload the P-states selection responsibility to the hardware for the most recent CPUs, being known as *Hardware P-states* (HWP). While such behavior can enhance the power efficiency and the performance of the CPU, the lack of fine-grained control of these states prevents the adoption of most of the available power estimations methods, which rely on an *a priori* calibration phase to build a static power model from the execution traces of a given workload run under variable frequencies.

D. Limitations & Opportunities

In the state of the art, most contributions require offline calibration and previous knowledge of relevant Performance Events for the power model which makes impractical the deployment in a highly heterogeneous infrastructure, like the Cloud. Because of the lack of documentation of performance events, and the constraints imposed by the HwPC slots, the automatic selection of relevant performance events remain an open challenge to support more adaptive power models that can adjust to changes in the workload or the context and keep optimizing the accuracy of power estimations. In SELFWATTS, we propose a new software-defined Power meter that leverages the state of the art in this area to automatically explore and select the relevant performance events at runtime with no offline calibration and optimizes the learned power models when necessary.

III. POWER MONITORING WITH SELFWATTS

A. Approach Overview

To learn power models that continuously deliver the best accuracy, no matter the workload and the deployment target, we propose an approach that explores the space of performance events incrementally and evaluates the impact of available events on the energy consumption of the monitored host. We propose to apply this approach at runtime, while the software-defined power meter keeps running to challenge the current power model with alternative performance events. This online learning approach aims to ensure the convergence of the power model towards the best combination of performance events that characterize the power consumption of a given workload and target architecture. While this approach may require some time to converge, we consider that the context of cloud computing assumes long-term monitoring that can accommodate such a training phase, given the timescale of virtual machine deployments.

As introduced in Figure 1, our approach introduces a feedback loop between the inference of a power model and the monitoring of performance events. By doing so, the monitoring becomes aware of the relevance of the selected events and can adjust the exploration of performance events accordingly.

More specifically, the monitoring component starts by selecting a subset of performance events $\langle e_1, \ldots, e_n \rangle \in E$ from the list of events E available on the target CPU (cf. Section III-D). The cardinality of the selected set of events depends on the number of HWPC slots that can be used concurrently. The monitoring component forwards metrics samples for the selected events to the inference component, which applies a supervised machine learning algorithm to

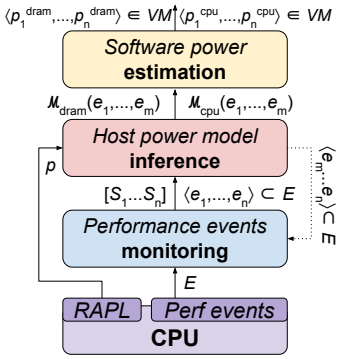

Fig. 1. Overview of the SELFWATTS approach.

establish a relationship between the energy consumption p retrieved from the RAPL interface and the selected events (cf. Section III-B). This model is further exploited by the estimation component to report on the power consumption of individual virtual machines or containers (cf. Section III-C). The inference component also reports on the list of irrelevant performance events, which can be used by the monitoring component to select another set of performances events among E. In this approach, performance events used by the power model are kept by the monitoring component, which only replaces the irrelevant performance events. By doing so, the power model is intended to converge towards a model that retains the performance events delivering the most accurate power estimations.

The following sections dive into the specific challenges of each component, starting from the host power model inference (cf. Section III-B), before explaining how software power estimation works (cf. Section III-C) and finally, how the performance events monitoring explores the space of available events (cf. Section III-D).

B. Host Power Model Inference

First, we consider that, for any hardware resource $res \in \{pkg, dram\}$ exposed by the RAPL interface, the associated power consumption p_{res}^{rapl} can be modelled as:

$$p_{res}^{rapl} = p_{res}^{static} + p_{res}^{dyn} \tag{1}$$

where p_{res}^{static} refers to the static power consumption of the monitored resource, and p_{res}^{dyn} reflects the dynamic power dissipated by the processor along the sampling period. By default, SELFWATTS consider p_{res}^{static} to be 0 and will spread the static consumption of the host across the active containers and virtual machines proportionally to theirs activity. However, other power accounting policies where the static consumption needs to be specifically handled can be implemented.

As previously introduced, the accuracy of a power model $\mathscr{M}_{res}^{f}$ strongly depends on *i)* the selection of relevant input features (performance events e_i) and *ii)* the acquisition of input samples that are evenly distributed along with the reference power consumption range. To better deal with the power features of hardware components, we group the input samples per operating frequency $f \in \mathscr{F}$, being the set of all frequencies operated by the hardware resource. Thus, we learn frequency-specific power models, aiming to converge automatically to a stable and precise representation over time. By tagging the samples along with the frequency operated by the processor, SELFWATTS ensures that the learned power models do not overfit the current context of execution, which may lead to inaccurate power models. The sampling tuples S_k^f, containing the raw sampled performance events, are grouped into memory as frequency layers $\mathscr{L}_{res}^{f} = [S_1^f, \ldots, S_n^f]$, which are the input features we maintain to build $\mathscr{M}_{res}^{f}$.

To classify the samples in the layer corresponding to the current frequency of the processor, SELFWATTS compute the average running frequency $\overline{f}$ as follows:

$$\overline{f} = f_{base} * \frac{\Delta \texttt{ APERF}}{\Delta \texttt{ MPERF}} \tag{2}$$

where f_{base} is the processor base frequency constant extracted from the field *Package Maximum Non-Turbo Ratio* of the `PLATFORM_INFO` *Model Specific Registers* (MSR). The `APERF` and `MPERF` variables are MSR-based counters that increment at the current and maximum frequencies, respectively. These counters are continuously updated, hence they report on a precise average frequency without consuming the limited HWPC slots. Interestingly, the performance power states, such as P-states and Turbo Boost, are covered by these counters as they act mainly on the frequency of the core to boost the performance. The idle optimization states (C-states) are also included, as they mainly reduce the average frequency of the core towards its *Max Efficiency Frequency* before being powered-down.

To filter out the irrelevant performance events for the power model, we compute a ranking of performance events using a *Recursive Feature Elimination* (RFE) and a cross-validated selection of the best number of features. This phase considers not only the raw input samples, but also transformed samples obtained from a fixed set of transformers, such as $Log, Exp, Sqrt, Cbrt, MinMaxScaler, StandardScaler, RobustScaler, Normalizer$, to boost the accuracy of the inferred models. The output of this phase is an ordered list of event and transformer combinations that can be used to infer a power model based on the subset of relevant performance events $\mathscr{E} = \langle e_1, \ldots, e_m \rangle \subset \langle e_1, \ldots, e_n \rangle \subset E$. From these filtered events, we can infer a frequency-specific power model $\mathscr{M}_{res}^{f} = [\gamma_1, \cdots, \gamma_m]$ that correlates, for a given frequency f, the dynamic power consumption ($\hat{p}_{res}^{dyn}$) to the raw samples for the frequency f that associated to set of relevant performance events $\mathscr{E}$, $\mathscr{L}_{res}^{f}(\mathscr{E})$:

$$\exists f \in \mathscr{F},\ \hat{p}_{res}^{dyn} = \mathscr{M}_{res}^{f} \cdot \mathscr{L}_{res}^{f}(\mathscr{E}) \tag{3}$$

In SELFWATTS, we learn $\mathscr{M}_{res}^{f}$ from a *Lasso* regression applied over the past k samples filtered by $\mathscr{E}$, $S_k^f = \langle p_{res}^{dyn}, e_1, \cdots, e_m \rangle$, with $p_{res}^{dyn} = p_{res}^{rapl} - p_{res}^{static}$.

To ensure that the power consumption of hosted virtual machines (or any application) is consistent with regards to the global power consumption of the host, we check that the

intercept belongs to the range $[0, \text{TDP}]$ where TDP refers to the *Thermal Design Power* of the CPU. By comparing $p_{res}^{dyn} + p_{res}^{static}$ with p_{res}^{rapl}, we can continuously estimate the error $\varepsilon_{res} =\mid p_{res}^{dyn} - \hat{p}_{res}^{dyn} \mid$ from estimated values in order to monitor the accuracy of the power model M_{res}^{f}. This estimation error is then stored in a sliding window of k samples to keep track of the accuracy of the model over time:

$$\tilde{\varepsilon}_{\text{res}}^{f} = med(\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n) \quad (4)$$

Whenever the median error $\tilde{\varepsilon}_{\text{res}}^{f}$ of the window exceeds a given threshold α set by the administrator, we assumes that a new power model requires to be inferred for the frequency f by reasoning over the latest input samples forwarded by the monitoring component.

C. Software Power Estimation

Given that we learn the host power model $\mathscr{M}_{res}^{f}$ from aggregated relevant performance events samples, $\mathscr{L}_{res}^{f} = \sum_{c \in C} \mathscr{L}_{res}^{f}(c)$, we can predict the power consumption of any container or virtual machine $c \in \mathscr{C}$ by applying the inferred power model $\mathscr{M}_{res}^{f}$ to the input samples associated to c, $\mathscr{L}_{res}^{f}(c)$:

$$\exists f \in \mathscr{F},\ \forall c \in \mathscr{C},\ \hat{p}_{res}^{dyn}(c) = \mathscr{M}_{res}^{f} \cdot \mathscr{L}_{res}^{f}(c)(\mathscr{E}) \quad (5)$$

Then, we distribute the value of the intercept i that is included in the estimate $\hat{p}_{res}^{dyn}(c)$ proportionally to the dynamic part of the consumption of c

$$\forall c \in \mathscr{C},\ \tilde{p}_{res}^{dyn}(c) = \hat{p}_{res}^{dyn}(c) - i \times (1 - \frac{\hat{p}_{res}^{dyn}(c) - i}{\hat{p}_{res}^{dyn} - i}) \quad (6)$$

In theory, one can expect that $\hat{p}_{res}^{dyn} \stackrel{!}{=} p_{res}^{dyn}$ if the model perfectly estimates the dynamic power consumption but, in practice, the predicted value may introduce an error $\varepsilon_{res} =\mid p_{res}^{dyn} - \hat{p}_{res}^{dyn} \mid$. Therefore, we cap the power consumption of any container c as:

$$\forall c \in \mathscr{C},\ \lceil \tilde{p}_{res}^{dyn}(c) \rceil = \frac{p_{res}^{dyn} \times \tilde{p}_{res}^{dyn}(c)}{\hat{p}_{res}^{dyn}} \quad (7)$$

to ensure that $p_{res}^{dyn} = \sum_{c \in C} \lceil \tilde{p}_{res}^{dyn}(c) \rceil$, thus avoiding potential outliers. Thanks to this approach, we can also report on the confidence interval of the power consumption of containers by scaling down the observed global error:

$$\forall c \in \mathscr{C},\ \varepsilon_{res}(c) = \frac{\tilde{p}_{res}^{dyn}(c)}{\hat{p}_{res}^{dyn}} \times \varepsilon_{res} \quad (8)$$

D. Performance Events Monitoring

SELFWATTS aims to explore the space of available performance events to monitor the most relevant set of events $\mathscr{E}$ that accurately model the power consumption of the host.

To do so, the monitoring component lists all the available performance events E that can be monitored from the target architecture together with the number of HWPC slots s that can be used for monitoring these performance events without triggering multiplexing effects, which seriously impact the accuracy of the input samples. Then, it randomly picks a set of s performance events $\langle e_1 \ldots e_n \rangle \subset E$ and configures the HWPC slots accordingly. The resulting set of input samples $S_k^{\bar{f}}$ is forwarded to the power model inference component to be used whenever a new power model requires to be learned.

When picking the set of events to be monitored, the events that are included in the current power model are kept in the new set, while the previous subset of events that were tagged as irrelevant by the inference component is replaced by another set of unexplored performance events taken from E.

One can note that, to speed up the convergence of power models and reduce the delay to produce accurate estimations, one can configure the monitoring component with a set of performance events that should be considered in priority. For example, this hint can be used to favour performance events, like *unhalted core cycles*, *unhalted ref cycles*, *instructions retired*, *llc misses/prefetch*, or *memory transactions cycles*, which are commonly adopted by the literature and then let SELFWATTS evaluate the relevance of these events in the deployment context, possibly identifying alternative performance events that better fit the power consumption of the target host.

IV. IMPLEMENTATION DETAILS

SELFWATTS builds on the POWERAPI toolkit[1] to implement the self-optimizing power modelling approach. Interestingly, POWERAPI was designed as a modular software system that can run atop a wide diversity of production environments. More specifically, SELFWATTS is an extension of SMARTWATTS [1] that introduces three key components: `Controller`, `Sensor`, and `Formula`. `Controller` and `Sensor` are covering the monitoring phase of SELFWATTS, while the inference and estimation phases are implemented by the `Formula`.

A. A Sensor to Monitor Performance Events

This component, developed in C, is a lightweight software daemon that uses *Hardware Performance Counters* (HWPC) to monitor a given set of performance events for all the Cgroups available on the host. We monitor Linux's Kernel *Control Groups* (Cgroups), as they are widely used by software container (Docker, LXC, Kubernetes) and virtual machine (libvirt) technologies, thus offering the adequate granularity to deliver software power estimations. The `Sensor`, therefore, periodically reports on samples of performance events per Cgroup, with a frequency that can be configured upon start ($\beta = 2\,\text{Hz}$ by default). This component represents the minimal requirement to obtain power estimations from a target architecture and is carefully implemented to limit its impact on hardware resources (CPU, DRAM) and co-located processes (containers, virtual machines).

B. A Controller to Explore Performance Events

This component, developed in C, is in charge of controlling the `Sensor` by configuring it with the appropriate set of performance events to monitor. The `Controller` uses the

[1] http://powerapi.org

Libpfm4 library[2] to detect the available *Performance Monitoring Unit* (PMU) of the target architecture, the number of HwPC slots and list the associated performance events. In SELFWATTS, the resulting set of performance events is randomly shuffled before starting the exploration. The `Controller` obtains the list of irrelevant performance events from the `Formula` and kills the active `Sensor` to replace it by another instance configured with a new selection of performance events. This new selection includes all the relevant performance events of the active power models (including CPU and DRAM power models) and completes the set with the next events consumed from the shuffled list. Once fully consumed, the list of available performance events is reset and shuffled with another random seed, thus resulting in a different combination of performance events to be explored by our approach.

C. A Formula to Optimize Power Models

As long as the median error of the active power models remains below the configured threshold ($\alpha = 5\,W$ by default), the `Formula` component delivers power estimation at the pace of forwarded samples (twice a second by default). If the median error exceeds this α threshold, then the `Formula` discards the active power model to infer a new power model from the new set of performance events. Given that the list of irrelevant events is forwarded to the `Controller` as soon as a new power model is computed, the `Controller` can anticipate by starting a new `Sensor` with a set of performance events. The list of relevant performance events will therefore be used to deliver power estimations as long as the power model is kept active, while the remaining performance events will be accumulated and consumed by a new power model will be requested, thus drastically reducing the delay to infer a new power model. The power model inference is implemented in Python and leverages *Scikit-learn*, which is the *de facto* standard Python library for general-purpose machine learning.[3]

D. Deployment of SELFWATTS

All the components of SELFWATTS are made to be deployed as Docker containers to ease their deployment and lifecycle management in container-based environments. The `Controller` embeds the `Sensor` and is deployed on all host machines. Optionally, one can use a MONGODB instance as a message queue to communicate input samples through a publish-subscribe pattern and as a *time series database* (TSDB) to store power estimations.

Figure 2 depicts an example of SELFWATTS configuration and compares it to the deployment of SMARTWATTS. In this configuration, the `Formula` component can be hosted by a remote virtual machine in charge of delivering the power estimations for all the monitored hosts and Cgroups. But, given the modularity of POWERAPI, SELFWATTS can also be deployed as a standalone solution where the components `Controller`/`Sensor` and `Formula` are co-located on the host machine. The latter deployment scheme is the one we consider in the following section to assess the accuracy and overhead of SELFWATTS in a "*worst-case*" configuration, which requires all the computations to be completed on the monitored host.

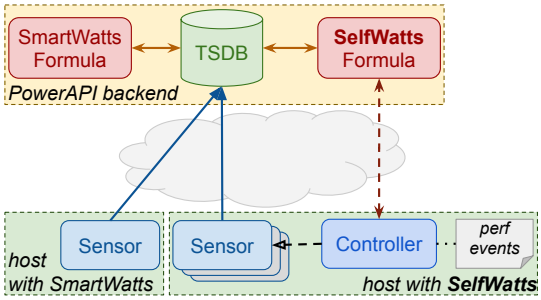

Fig. 2. Deployment of SELFWATTS, compared to SMARTWATTS.

V. EMPIRICAL EVALUATION

This section assesses the accuracy and the efficiency of SELFWATTS to select relevant performance events and to estimate the power consumption of hosted virtual machines with accuracy. More specifically, this section addresses the following research questions:

- **RQ 1**: How does SELFWATTS compare to SMARTWATTS in terms of accuracy?
- **RQ 2**: What is the runtime overhead of SELFWATTS?
- **RQ 3**: How does SELFWATTS adapt to different target architectures?

A. Evaluation Methodology

We follow the experimental guidelines reported by [13] to enforce the quality of our empirical results. For the sake of reproducible research, SELFWATTS, the necessary tools, deployment scripts, and resulting datasets are open-source and publicly available on GitHub.[4]

1) Testbeds: Our setups can be reproduced on the GRID5000 testbed infrastructure,[5] which provides large clusters of machines for experiment-driven research. To assess the versatility of our approach, we consider several heterogeneous processor architectures that exhibit different characteristics and combinations of power-aware features, as reported in Table I.

TABLE I
TESTBED HARDWARE SETTINGS

Model	Dell PowerEdge C6420	Dell PowerEdge R730	Dell PowerEdge R630
CPU	Intel Xeon Gold 6130	Intel Xeon E5-2650 v4	Intel Xeon E5-2630 v3
Generation	Skylake	Broadwell	Haswell
Cores per-socket	16	12	8
Thread(s) per-core	32	24	16
Socket(s)	2	2	2
TDP	125 W	105 W	85 W
Memory	192 GiB	128 GiB	128 GiB
# Perf. Events	257	260	265
# HwPC slots	3 fixed / 4 generic	3 fixed / 4 generic	3 fixed / 4 generic

[2]https://github.com/wcohen/libpfm4
[3]https://scikit-learn.org
[4]https://github.com/powerapi-ng
[5]https://www.grid5000.fr

Both the host and virtual machines are using the Ubuntu 20.04.1 LTS Linux distribution with the `5.4.0-53-generic` Kernel version, where only a minimal set of daemons are running in background. All the selected workloads run inside QEMU[6] virtual machines managed by libvirt.[7]

2) Workloads: Our workloads are based on standard benchmarks, like STRESS NG[8] and NASA's *NAS Parallel Benchmarks* (NPB 3) [14], to highlight the benefits of our approach. The experiment workload is split into two phases, SEQUENTIAL where these benchmarks will run one after the other, and PARALLEL where all the benchmarks will run concurrently.

3) Power meters: In all our experiments, we configure the `Controller`/`Sensor` components SELFWATTS to report on power estimations twice a second ($\beta = 2\,\text{Hz}$), and the FORMULA component with an error threshold of $\alpha = 5\,W$, which are the default parameters of SELFWATTS.

We evaluate and compare the following configurations:

1) SmartWatts refers to the configuration the SMARTWATTS power meter as published in [1],
2) SELFWATTS (default) to the configuration of SELFWATTS with default settings,
3) SELFWATTS *with fixed events* starts SELFWATTS with the following `x86` performance events: `UNHALTED_REFERENCE_CYCLES`, `UNHALTED_CORE_CYCLES`, `INSTRUCTION_RETIRED`, which consumes the 3 fixed HWPC slots and are commonly considered by the state of the art.

B. Experimental Results

We start by reporting in Figures 3 and 4 on the power estimations reported by the default configuration of SELFWATTS when running our sequential and parallel workloads, respectively. The cumulated execution of these workloads lasts for 30 minutes on a Dell PowerEdge C6420 machine and both figures report on 3 classes of power estimations. First, the kernel and system power profiles reflect the Linux kernel and all the operating system activities, respectively. One can observe that the power consumption of the kernel and system layers remains low in general, by reaching up to 2 W in the sequential phase, but may go above 25 W when dealing with the UDP stress in the parallel phase.

Then, the selfwatts-controller and selfwatts-formula power profiles illustrate the activity of SELFWATTS along with the benchmark. This power profile highlights periods where SELFWATTS invest energy to find a new power model by exploring alternative performance events before reaching more stable period with a reduced power consumption that reflect the exploitation of a stable[9] and accurate power model.

Finally, the lower part of both figures depicts the individual power profiles of NPB and STRESS NG workloads, which are isolated in dedicated virtual machines. One can observe that, no matter the workload and the number of involved cores, SELFWATTS keeps delivering power estimations with accuracy and low overhead. We further investigate these claims in the remainder of this section.

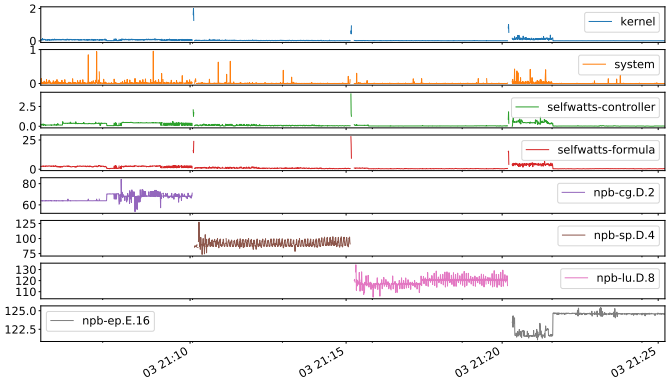

Fig. 3. Power estimations per VM for SEQUENTIAL phase

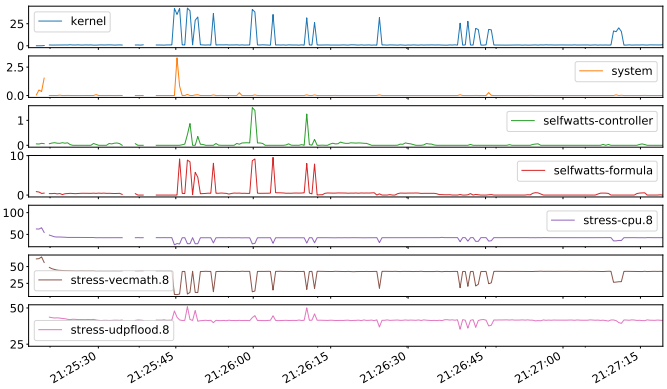

Fig. 4. Power estimations per VM for PARALLEL phase

1) Estimation accuracy: To assess the accuracy of SELFWATTS, and answer **RQ 1**, we start by reporting on the error ε of the 3 configurations under study. The statistics reported in Figure 5 show that SELFWATTS succeeds to compete with SmartWatts in terms of accuracy by reaching the same error on average (2 W for the host estimation), far below the error threshold of $5\,W$ we used for all the configurations. Yet, SELFWATTS reports on more occurrences of larger errors, as its exploration phase may lead to inaccurate estimations for short periods, compared to SmartWatts which boots with an accurate model and never explores alternative performance events, essentially adjusting the coefficients of the power model $\mathscr{M}$ by computing a new *Ridge* regression, while SELFWATTS combines RSE and a *Lasso* regression to perform on-the-fly performance events selection and power model optimization. Our approach, therefore, goes one step beyond SmartWatts by adopting a more dynamic power modeling approach that takes the freedom to consider alternative performance events to optimize the power model.

To further investigate the errors reported by SELFWATTS,

[6]https://www.qemu.org
[7]https://libvirt.org
[8]https://launchpad.net/stress-ng
[9]a power model is considered as stable as long as it keeps estimating host power consumption with a median error $\tilde{\varepsilon}$ below the configured threshold α.

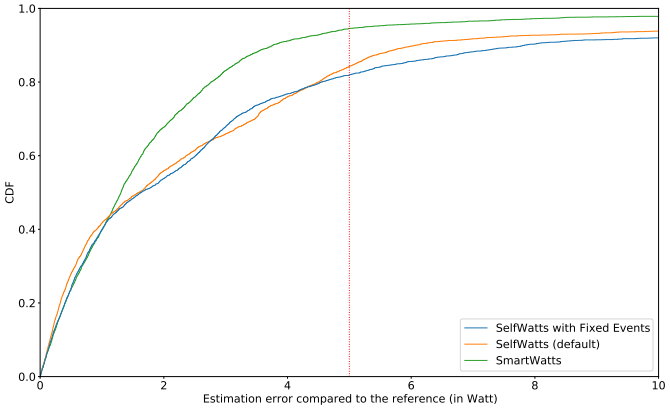

Fig. 5. CDF of estimation errors ε for SELFWATTS and SmartWatts

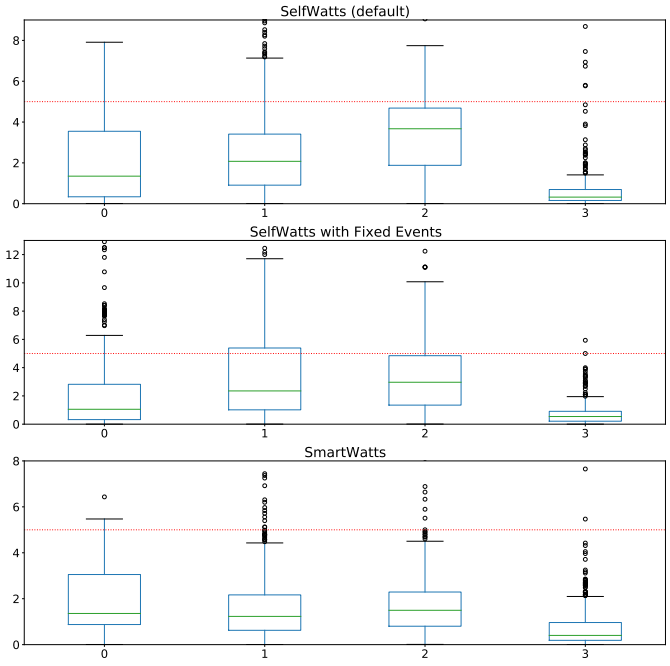

Fig. 6. Estimation errors ε per application involved in the SEQUENTIAL phase

we compare the evolution of this error ε along with phases and steps of the workloads. We, therefore, split the SEQUENTIAL phase into 4 consecutive steps, which are aligned with the 4 NPB applications that are executed during this phase (cf. Figure 6). One can observe that, no matter the applications, both configurations of SELFWATTS reach a similar accuracy compared to SmartWatts.

These results can be further confirmed with the PARALLEL phase of our workload in Figure 7. Although SELFWATTS can be perceived as less accurate than SmartWatts, our manual investigations revealed that SmartWatts entered the PARALLEL phase with a power model that exhibited a high error ratio and triggered the inference of a much more accurate power model, while the power models exploited by the two configurations of SELFWATTS turned out to be a stable, but a bit less accurate (still far below the configured error threshold).

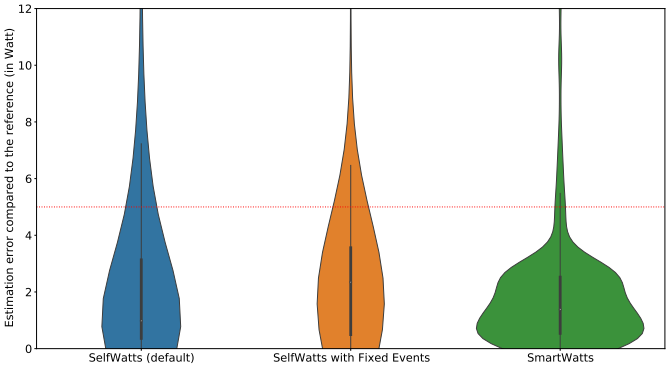

Fig. 7. Estimation errors ε for the PARALLEL phase

TABLE II
POWER CONSUMPTIONS OF THE CONTROLLER/SENSOR COMPONENT

Configuration	Avg	Std	Energy
SELFWATTS (default)	0.26 W	0.75 W	600 J
SELFWATTS *with fixed events*	0.20 W	0.44 W	465 J
SmartWatts	0.16 W	0.20 W	379 J

2) Runtime overhead: To answer **RQ 2**, we then explore the power consumption of SELFWATTS to investigate the overhead imposed by our solution on the monitored host. Tables II and III more specifically report on the average power consumption of the `Controller`/`Sensor` and `Formula` components we implemented. The `Formula` component runs with the PYPY runtime for Python, version 7.3.3.[10] Interestingly, as both `Controller`/`Sensor` and `Formula` components are deployed as DOCKER containers, SELFWATTS can monitor and deliver fine-grained power estimations of its components in addition to the monitored VMs.

TABLE III
POWER CONSUMPTIONS OF THE FORMULA COMPONENT

Configuration	Avg	Std	Energy
SELFWATTS (default)	0.64 W	1.53 W	1,458 J
SELFWATTS *with fixed events*	0.42 W	0.52 W	960 J
SmartWatts	0.33 W	1.65 W	817 J

We also study the evolution of the power consumption of SELFWATTS when monitoring an increasing number of hosted virtual machines on a single node. Thus, Figures 8 and 9 report on the power consumptions of the DRAMand CPU hardware resources for both components when monitoring a growing number of hosted virtual machines that ranges from 0 to 200, by starting 10 new VMs every 10 seconds (vertical red lines refers to increments of 40 VMs). The consumption spikes observed in Figure 8 refer to periodic adjustments of the coefficients of the power model $\mathscr{M}$ when additional VMs are started, while the first consumption spike observed in

[10]https://www.pypy.org

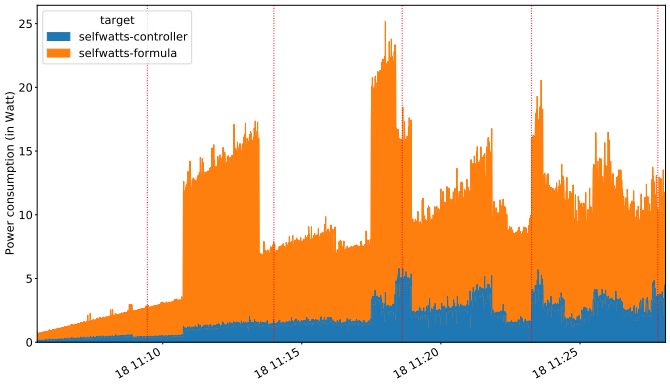

Fig. 8. CPU power consumption of the `Controller/Sensor` and `Formula` components

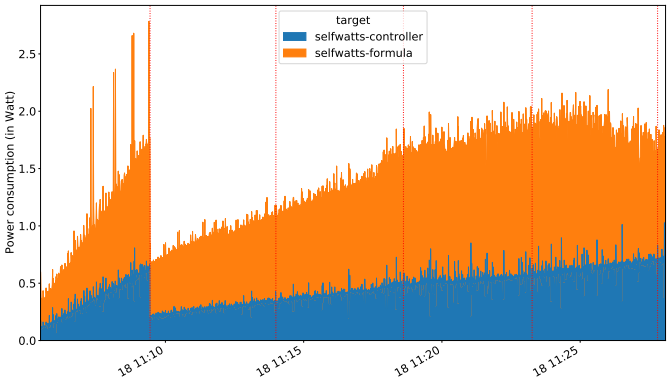

Fig. 9. DRAM power consumption of the `Controller/Sensor` and `Formula` components

Figure 8 refers to a change of the performance events used by the power model $\mathscr{M}$. This is due to the fact that the node moves from 0 to 40 hosted VMs, which drastically change the execution context, but demonstrates that SELFWATTS succeeds in identifying In the latter case, one can observe that SELFWATTS succeeds to free the memory associated to irrelevant performance events.

Interestingly, when reaching 200 VMs, the default configuration of SELFWATTS consumes $10\,W$ and $1.5\,W$ for the CPU and DRAM resources, on average, which represents a cost per VM of $0.06\,W$. Among the factors that can contribute to further reduce this overhead, one can mention the exploitation of fixed events (cf. Tables II & III), the reduction of the monitoring rate β, or increasing the error threshold α.

With regards to the power consumption profiles of monitored applications, ranging from $25\,W$ to $125\,W$ (cf. Figures 3 & 4), we can answer **RQ 2** and conclude that SELFWATTS offers a lightweight solution to monitor the power consumption of virtual machines and containers.

3) Self-optimization: Finally, to answer **RQ 3**, we deployed the default configuration of SELFWATTS on three target architectures that exhibit a different set of performance events (cf. Table I). In this mode, SELFWATTS offers a zero-configuration solution to automatically converge towards accurate power

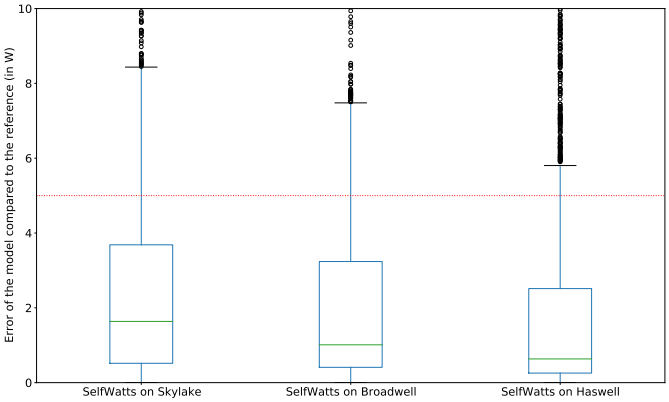

Fig. 10. Estimation errors ε for SELFWATTS on across different CPUs

models for any target architecture, as reported in Figure 10. Our results show in particular that, no matter the target architecture, SELFWATTS succeeds to estimate the power consumption of the host machine with high accuracy, which contributes to estimate the power consumption at the scale of virtual machines and containers with higher confidence. We believe that this accuracy is particularly critical when further estimating the power consumption of guest software systems, as proposed by [10].

C. Lessons Learned & Perspectives

Beyond the high accuracy and the low overhead exhibited by SELFWATTS, we would like to share the lessons we learned from designing and implementing such a self-optimizing middleware solution.

First, the monitoring of performance events requires carefully considering the pitfalls related to the limited number of HWPC slots. While SELFWATTS detects the number of available HWPC slots that can be used to explore the relevant performance events, nothing prevents another co-located software system to monitor other performance events and incidentally impact the accuracy of SELFWATTS by triggering multiplexing at the level of the hardware performance counters.

Surprisingly, the randomization process introduced by the `Controller` component of SELFWATTS conducted to the inference of power models exploiting performance events that are different from the ones commonly identified by the community, yet achieving similar accuracy. Furthermore, due to the stochastic nature of the monitored environment, nothing prevents SELFWATTS to exploit power models based on a different set of relevant performance events when running the same workload on the same target architecture. Nevertheless, both our accuracy and overhead investigations show that the lack of convergence towards a single power model is not a limitation of our solution, contrary to state-of-the-art claims about the fact only a limited number of performance events can accurately capture the power consumption of a target architecture.

Unfortunately, the introduction of input feature transformers in the `Formula` component of SELFWATTS did not lead

to expected results, as most of the inferred power models we manually analyzed rather exploit the raw input samples after completing the *Lasso* regression. We, nonetheless, believe that this negative result deserves to be mentioned as part of the lessons we learned to open discussions for the relevance of such methods commonly adopted in machine learning in the specific-case of online supervised training based on performance events.

Through the definition of an error threshold, our approach enforces a trade-off between the stability of the inferred power models and their accuracy (cf. Figure 7). This parameter α, therefore, indirectly controls the decision process of SELFWATTS to balance the exploration of a more accurate power model versus the exploitation of an acceptable power model, yet not optimal. While this problem is commonly known in reinforcement learning communities, we believe that it is particularly critical as our experimentation have shown that the exploration phases inevitably induce a power consumption overhead.

Finally, while SELFWATTS ambitions to support any target architecture, our experiments with an AMD EPYC 7301 (32 cores / 64 threads) has shown that, although the RAPL interface is being supported on latest Linux kernel versions, *i)* the RAPL support for the DRAM is still lacking and *ii)* the support for performance events remains immature compared to Intel-based processors. We nonetheless believe that future developments of AMD-related libraries will fix this limit in a near future.

Finally, beyond the specific case of power modeling we explore in the context of this paper, we believe that the proposed architecture could also benefit to other case studies that require to downscale ground truth observations (here RAPL measurements) to the scale of individual Cgroups, by automatically correlating causal connections between global observations and more fine-grained activities. For example, the identification and monitoring of side-channel attacks in the domain of security could leverage our contribution.

VI. CONCLUSION

Power consumption is a critical concern in modern distributed computing infrastructures, from HPC to data centers, which more and more aim to implement sustainable solutions to cope with environmental challenges raised by the massive deployment of software services. While current practices leverage tools to monitor the power consumption at a coarse granularity (*e.g.*, nodes, sockets), the literature still fails to propose generic power models, which can be easily deployed and used to estimate the power consumption of software artifacts in production with accuracy. This failure can be explained not only by the prohibitive cost of some solutions and models deployed on monitored hosts, but also the consideration of a static set of input features (*e.g.*, performance events) that may not be available on a specific target architecture, thus compromising the deployment of the monitoring solution.

In this paper, we reported on a novel zero-configuration power meter, named SELFWATTS, that automatically selects the relevant performance events and continuously self-optimize the power models that can be used to deliver real-time power estimations with accuracy. Interestingly, we demonstrate that, no matter the target architecture, SELFWATTS does not require a prohibitive offline calibration phase to maintain a power model that can report the power consumption of software containers or *virtual machines* (VM). The experimental results we conducted highlights that SELFWATTS can estimate the power consumption of an unknown host with an average error of $2\,W$ ($1.6\,\%$ of the TDP) for a monitoring cost of $0.06\,W$ per monitored VM.

Thanks to SELFWATTS, system administrators and developers can analyze the power consumption of individual software containers and virtual machines to detect anomalies and/or identify optimizations for their distributed systems. In particular, instead of addressing performance bottlenecks by provisioning additional hardware resources, we believe that SELFWATTS can contribute to increase the energy efficiency of distributed software systems at large.

The code of SELFWATTS is freely available from http://powerapi.org and can be easily deployed in production as a Docker container.

REFERENCES

[1] G. Fieni, R. Rouvoy, and L. Seinturier, "Smartwatts: Self-calibrating software-defined power meter for containers," in CCGrid, 2020, pp. 479–488.

[2] M. Colmant, P. Felber, R. Rouvoy, and L. Seinturier, "WattsKit: Software-Defined Power Monitoring of Distributed Systems," in CCGrid. IEEE / ACM, 2017, pp. 514–523.

[3] M. Kurpicz, A. Orgerie, and A. Sobe, "How much does a VM cost? energy-proportional accounting in vm-based environments," in Euromicro. IEEE, 2016, pp. 651–658.

[4] M. Colmant, R. Rouvoy, M. Kurpicz, A. Sobe, P. Felber, and L. Seinturier, "The next 700 CPU power models," JSS, vol. 144, pp. 382–396, 2018.

[5] D. Versick, I. Waßmann, and D. Tavangarian, "Power consumption estimation of cpu and peripheral components in virtual machines," SIGAPP Appl. Comput. Rev., vol. 13, no. 3, p. 1725, 2013.

[6] R. Bertran, M. González, X. Martorell, N. Navarro, and E. Ayguadé, "Decomposable and responsive power models for multicore processors using performance counters," in ICS. ACM, 2010, pp. 147–158.

[7] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan, "Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge," Micro, 2012.

[8] S. Desrochers, C. Paradis, and V. M. Weaver, "A validation of DRAM RAPL power measurements," in MEMSYS, 2016, pp. 455–470.

[9] K. Shen, A. Shriraman, S. Dwarkadas, X. Zhang, and Z. Chen, "Power containers: an OS facility for fine-grained power and energy management on multicore servers," in ASPLOS. ACM, 2013, pp. 65–76.

[10] M. Colmant, M. Kurpicz, P. Felber, L. Huertas, R. Rouvoy, and A. Sobe, "Process-level power estimation in vm-based systems," in EuroSys. ACM, 2015, pp. 1–14.

[11] M. LeBeane, J. H. Ryoo, R. Panda, and L. K. John, "Wattwatcher: Fine-grained power estimation for emerging workloads," in SBAC-PAD, 2015.

[12] R. Zamani and A. Afsahi, "A study of hardware performance monitoring counter selection in power modeling of computing systems," in IGCC. IEEE Computer Society, 2012, pp. 1–10.

[13] E. van der Kouwe, D. Andriesse, H. Bos, C. Giuffrida, and G. Heiser, "Benchmarking Crimes: An Emerging Threat in Systems Security," CoRR, vol. abs/1801.02381, 2018.

[14] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, and R. S. Schreiber, "The NAS parallel benchmarks," IJHPCA, 1991.